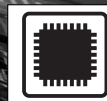
初歩からのHDLテストベンチ

第2回　クロックを含むテストベンチを書く

安岡貴志

※本連載記事の第1回は，本誌2007年5月号，pp.70-79の特集1 第4章「テストベンチの書き方を身に付ける」として掲載されました．

HDLで回路を記述できるようになったばかりで，これからテストベンチを書こうとしてる方を対象とした連載の第２回です．前回（本誌2007年５月号，pp.70-79）は，検証対象の回路のポートはすべて１ビットでした．今回は幅を持つ信号を扱います．　（筆者）

あなたは日常生活では，10進数[注1]に慣れ親しんでいると思いますが，ハードウェアの設計では2進数や16進数を使う場面が非常に多くなります．なぜならハードウェアの設計では信号の幅を意識しなければならないためです．2進数や16進数を使うと，ビットごとの信号の状態（‘0’か‘1’か）を捉えやすくなります．

表1に10進数，2進数，16進数の表現をまとめます．16進数の優れた点は，1けたが10進数以上の情報量を持ち，かつ各ビットの状態が明確になる点です．

例えば，2進数で11000101という値があるとします．これは10進数で197と表せますが，8ビット中の各ビット状態を把握するのは困難です．これに対して16進数であればC5と表すことができます．16進数の1けたは2進数の4けたに相当するので，**表1**を参照することにより，容易に各ビットの状態を把握できます．

表1　10進数，2進数，16進数の表現

10進数	2進数	16進数	10進数	2進数	16進数
0	0	0	8	1000	8
1	1	1	9	1001	9
2	10	2	10	1010	A
3	11	3	11	1011	B
4	100	4	12	1100	C
5	101	5	13	1101	D
6	110	6	14	1110	E
7	111	7	15	1111	F

1．幅のある信号の表記

4ビットの信号CNT4と，その信号を1ビットごとに分岐させた信号の様子を**図1**に示します．最下位ビットを0ビット目，最上位ビットを3ビット目の信号と表しています．各ビットのVerilog HDLとVHDLによる表記を**図1（a）**に示します．また，信号CNT4とその各ビットの信号の変化の例を示したのが**図1（b）**です．

Verilog HDL

リスト1（a）は，幅のある信号をVerilog HDLで宣言するときの書式です．

データ型は，`reg`もしくは`wire`となります．最上位ビットと最下位ビットの位置には，それぞれ整数が入ります[注2]．同じデータ型，ビット幅の信号であれば，“,”で区切って複数の信号を1行で宣言することができます．

リスト1（b）は幅のある信号の宣言の例です．

VHDL

リスト2（a）は，幅のある信号をVHDLで宣言するときの書式です．

注1：N進数のNのことを基数と言う．10進数の基数は10，16進数の基数は16である．
注2：ビットの数値は，最上位ビット＞最下位ビット，最下位ビットは0とするのが一般的である．

KeyWord　テストベンチ，Verilog HDL，VHDL，2進数，16進数，クロック，イネーブル付き12進カウンタ，レーシング

最上位ビット
4ビットの信号
CNT4
4
3ビット目の信号
2ビット目の信号
1ビット目の信号
0ビット目の信号
最下位ビット

Verilog HDLの表記	VHDLの表記
CNT4[3]	CNT4(3)
CNT4[2]	CNT4(2)
CNT4[1]	CNT4(1)
CNT4[0]	CNT4(0)

(a) 信号の表現

CNT4　3　C　A
CNT4[3]　CNT4(3)
CNT4[2]　CNT4(2)
CNT4[1]　CNT4(1)
CNT4[0]　CNT4(0)
ビットごとの切り出し
16進数表現

(b) タイミング・チャート

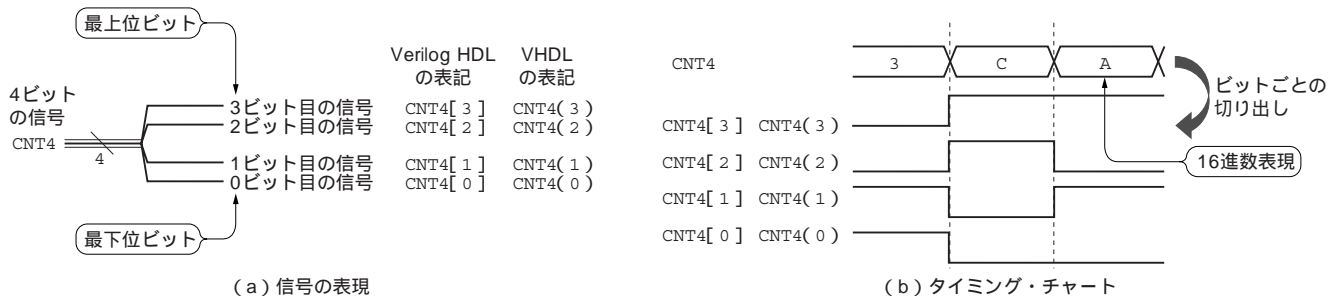



図1　幅のある信号の表記
(a)では，4ビットの信号と1ビットごとの表記を示している．また(b)は，実際の信号の変化の様子を示している．

リスト1　Verilog HDLによる幅のある信号の宣言

```
データ型 [最上位ビット:最下位ビット] 信号名;
```

(a) 書式

```
reg [3:0] CNT4;
wire [1:0] LS,HS;
```

(b) 記述例

リスト2　VHDLによる幅のある信号の宣言

```
signal 信号名 : std_logic_vector(最上位ビット downto 最下位ビット);
```

(a) 書式

```
signal CNT4 : std_logic_vector(3 downto 0);
signal SL,HS : std_logic_vector(1 downto 0);
```

(b) 記述例

(a) ブロック図

- ●回路の名前はcounterとする．
- ●カウント値は，出力ポートCNT4から出力される．カウント値は0から11までの値をとり，ビット幅は4ビット．
- ●非同期リセットが付いている．入力ポートRST_Xに接続された信号が‘0’(Lレベル)になると，カウント値は0に戻る．
- ●(RST_Xが1で)入力ポートCOUNTONに接続されている信号が‘1’のとき，クロック(入力ポートCLKに接続された信号)の立ち上がりごとに，カウント値は+1される(カウント値が11のときは0に戻る)．
- ●(RST_Xが1で)入力ポートCOUNTONに接続されている信号が‘0のとき，カウント値は保存される．

(b) 仕様

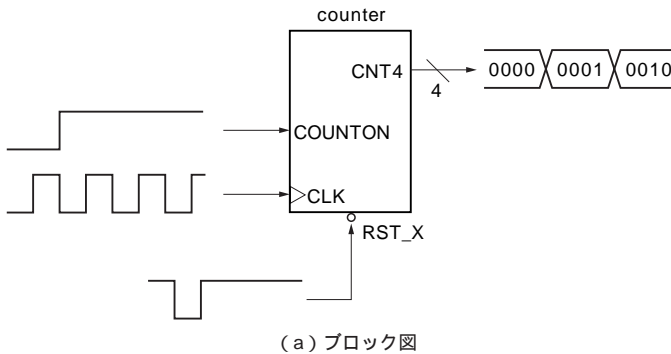



図2　イネーブル付きの12進カウンタの仕様
組み合わせ回路と異なりクロックが必要．

最上位ビットと最下位ビットの位置には，それぞれ整数が入ります[注3]．同じデータ型，ビット幅の信号であれば，“，”で区切って複数の信号を1行で宣言することができます．

リスト2(b)は幅のある信号の宣言の例です．

2. クロックを含むテストベンチの注意点

前回は，組み合わせ回路のテストベンチを元に，テストベンチの書き方の基本を解説しました．このため，前回のテストベンチにはクロックの記述がありませんでした．

クロックを含むテスト対象の回路(順序回路)は，組み合わせ回路と違い独特の注意が必要です．

今回はクロックを含む回路のテストベンチを作成します．検証対象の回路は，イネーブル付き[注4]の12進カウンタにします．仕様(機能)を**図2**に示します[注5]．

注3：ビットの数値は，downtoを使うときには最上位ビット＞最下位ビットでなくてはならない．最下位ビットは0が一般的である．
注4：イネーブルとは，なんらかの機能を許可する信号である．今回であればイネーブル信号COUNTONが‘1’のとき，カウント機能が許可になり，カウントを実行する．
注5：前回も説明したように，実際の開発ではこの規模でテストをすることはない．クロックを含む回路のテストベンチの要点を明確にするために，あえて小規模な回路にしている．

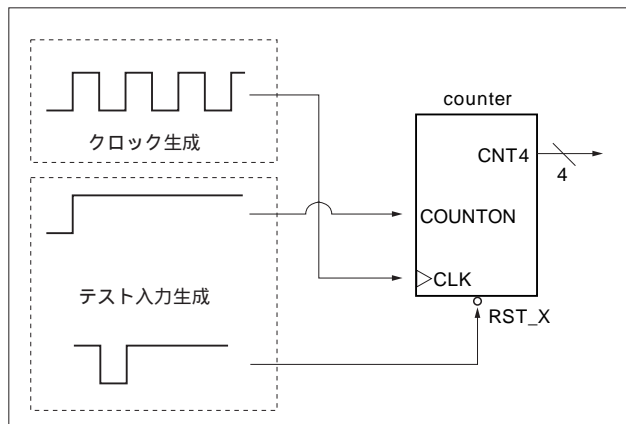



図3　イネーブル付きの12進カウンタのテストベンチ
クロックとテスト入力を生成する．

リスト3　Verilog HDLによるクロックの記述例

```
reg         CLK;

always begin
            CLK = 1'b1;
  #50       CLK = 1'b0;
  #50;
end
```

● クロックを作る

前回の組み合わせ回路のテストベンチと大きく違う点として，クロックの存在があります．クロックは，一定周期で‘0’(Lレベル)と‘1’(Hレベル)を繰り返す信号です．

今回のテストベンチは**図3**のようになります．

Verilog HDL

リスト3は，Verilog HDLでCLKというクロックを作ったものです．CLKは，50nsで‘0’(Lレベル)，‘1’(Hレベル)を繰り返します注6．CLKの周期は100nsになります．

クロックは，ほかのテスト入力とは別のブロック(`always`文)で作ります(下掲のコラム「クロックの記述」を参照)．Verilog HDLでは別々のブロック(`always`文や`initial`文など)は，並列に(並行して)動きます．

注6：本稿では説明を簡単にするために，Verilog HDLにおける時間単位がnsである前提で説明する．時間単位の初期設定はシミュレータにより異なる．

コラム　クロックの記述

Verilog HDL

前回解説したテスト入力と同じ`initial`文の中でクロックを作ろうとすると，**リストA**のようになってしまいます．これでは非常に大変ですね．そこで，クロックは同じ動作を永久に繰り返す`always`文を用いて作ります．

あなたが回路設計で使った`always`文では，**リストB(a)**のように，@とそれに続くかっこの中に信号が書かれていたと思います．これらの信号をセンシティビティ(センシティビティ・リスト)といい，その信号のどれかが変化すると，`always`文の中の式が実行されていました．

これに対し，クロックを記述した**リストB(b)**を見ると，@とそれに続くかっこがありません．この場合，`always`文の中の式は永久に実行され続けます．

クロックでは‘1’の代入と，‘0’の代入を一定の周期で永久に繰り返せばいいので，センシティビティなしの`always`文を使います．

リストA　initial文によるクロックの記述例

```
initial begin
        RST_X =1'b1; COUNTON = 1'b0;
        CLK = 1'b1;
  #50   RST_X   = 1'b0; CLK = 1'b0;
  #50   CLK = 1'b1;
  #50   RST_X   = 1'b1; CLK = 1'b0;
  #50   CLK = 1'b1;
  #50   CLK = 1'b0;
  #50   COUNTON = 1'b1; CLK = 1'b1;
  #50   CLK = 1'b0;
  #50   CLK = 1'b1;
      ・
      ・
      ・
  #50   $finish();
end
```


リストB　always文の使い方

```
always @(A or B or C) begin
   ・
   ・
   ・
end
```


(a) 回路のalways文

```
always begin
            CLK = 1'b1;
  #50       CLK = 1'b0;
  #50;
end
```


(b) クロックのalways文

VHDL

リスト4はVHDLでCLKというクロックを作ったものです．CLKは今回50nsで'0'(Lレベル)，'1'(Hレベル)を繰り返します．CLKの周期は100nsになります．

クロックは，ほかのテスト入力とは別のブロック(process文)で作ります(下掲のコラム「クロックの記述」を参照)．VHDLでは別々のブロック(process文など)は，並列に(並行して)動きます．

● レーシングに注意

テストベンチを作り始める前に，レーシングの問題を説明します．これはクロックの付いた(フリップフロップを含む)回路の検証をする上で，非常に重大な問題なので，最初に知っておく必要があります．

図3では，検証対象の回路のテスト入力として，入力ポートCLK，RST_X，COUNTON(に接続されている信号)が与えられています．ここで，CNT4の出力波形として，**図4**のように2通り考えられますが，どちらが正しいのでしょうか．

ここで注意しなければいけない点は，CLK(クロック)の立ち上がりと，COUNTONの変化タイミングが同一だということです．クロックの立ち上がりで，COUNTONが'0'だとすると上の波形が正しいということになります．逆に'1'だとすると下の波形が正しいことになります．これがレーシングと言われる状況です．

実はVerilog HDL，VHDLともにクロック・エッジで変化した信号の判定に関して規定がないため，シミュレータ次第となり，どちらもありえるということになります(p.120のコラム「リセット前のフリップフロップの値」を参照)．これではシミュレータを変えた途端，検証結果が違ってくるというような事態が発生し，大変不便です．

そこで，クロックの付いた(フリップフロップを含む)回路(順序回路)のテストベンチでは，テスト入力の変化点はクロックのエッジ(立ち上がりで回路が動作する場合，クロックの立ち上がり)を避けるのがセオリとなります．

図5は，テスト入力の変化点をクロック・エッジからずらした例です．クロックの付いた回路のテストベンチはこのように作ります．

リスト4　VHDLによるクロックの記述例

```
宣言部分
  signal CLK : std_logic;

  begin
機能部分
  process begin
    CLK <= '1'; wait for 50 ns;
    CLK <= '0'; wait for 50 ns;
  end process;
```

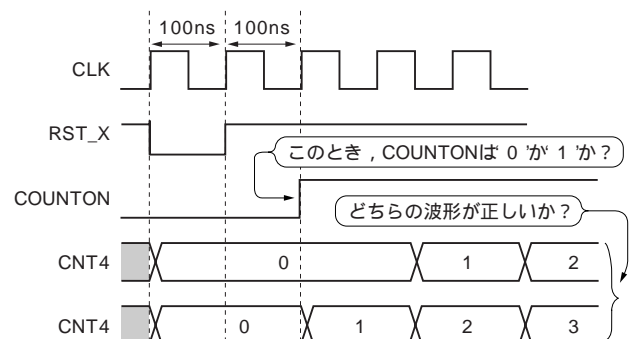



図4　レーシング

3クロック目の立ち上がりにおいて，COUNTONが'0'と判定されるとCNT4は上の波形に，COUNTONが'1'と判定されるとCNT4は下の波形になる．

VHDL

前回解説したテスト入力と同じprocess文の中でクロックを作ろうとすると，**リストC**のようになってしまいます．これでは非常に大変ですね．そこで，クロックは同じ動作を永久に繰り返す別のprocess文を用いて作ります．

クロックでは'1'の代入と'0'の代入を一定の周期で永久に続ければいいので，シミュレーションを停止する式のないprocess文を使います．

リストC　process文によるクロックの記述例

```
process begin
  RST_X <='1'; COUNTON <= '0'; CLK <= '1';
  wait for 50 ns; RST_X <= '0'; CLK <= '0';
  wait for 50 ns; CLK <= '1';
  wait for 50 ns; RST_X <= '1'; CLK <= '0';
  wait for 50 ns; CLK <= '1';
  wait for 50 ns; CLK <= '0';
  wait for 50 ns; COUNTON <= '1'; CLK <= '1';
  wait for 50 ns; CLK <= '0';
      ・
      ・   シミュレーション終了まで，50nsごとに
      ・   CLKに"0"と"1"を代入し続ける

  wait for 50 ns; assert false;
end process;
```

3．検証仕様とテスト入力の記述

● 検証のポイントを絞る

今回の回路を見ると，仕様として以下の項目が挙げられます．

- 非同期リセット
- COUNTONによるカウントの許可/停止
- カウントアップ

前回の回路で確認しなければならない内容は，たったの4状態しかありませんでした．これは入力信号の組み合わせが4通りしかなく，出力信号の状態が入力信号の組み合わせと1対1の組み合わせ回路（ANDゲート）だったからです．

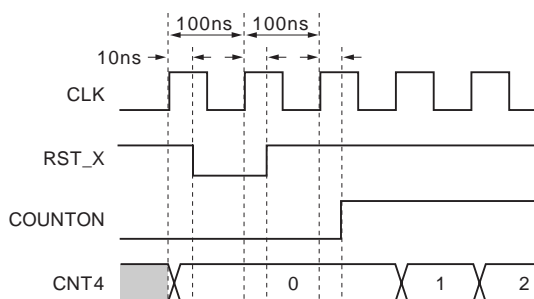



図5 テスト入力の変化点をクロック・エッジからずらす
テスト入力の変化点とクロックの立ち上がりが重なることがないので，‘0’か‘1’かが明確に決まる．

これに対して今回は順序回路であり，入力信号の状態がまったく同じであっても，前サイクル（直近のクロックの立ち上がり前）の回路の状態によって，現在の回路の状態は変わります．

例えばRST_Xが1，COUNTONが1で，前サイクルのCNT4が4であれば，CLKの立ち上がりでCNT4は5になります．同じ条件で，前サイクルのCNT4が5であれば，CLKの立ち上がりでCNT4は6になります．

もし今回，可能性のあるすべての状態を確認しようとすると非常に大変です．**図6**のように48通りにもなります．

それでは，最低限確認しなければならない項目に絞るとすると，どのような項目が残るでしょうか．必須項目を絞ると次の**図7**のような内容になります（p.122のコラム「検証仕様の洗い出し」を参照）．

これだけの項目を見ることができれば，100％とはいえないまでも，ほとんどの不具合は発見できます．このテスト入力の例を**図8**に示します．

● テスト入力の作成

今回，順序回路（記憶素子を含む回路）のテストベンチを作成するわけですが，クロックの記述とレーシング対策以外は前回の組み合わせ回路（記憶素子を含まない回路）のテ

コラム　リセット前のフリップフロップの値

現実のフリップフロップは，電源投入から最初のリセットまでは，“1”になっているが“0”になっているか分かりません．HDLではこの状態を表現する値があります（**図A**）．

Verilog HDL

Verilog HDLではこの状態を‘X’で表します．これは不定と呼ばれ，‘1’か‘0’か分からない状態を表します．

シミュレーション結果の波形を観察しているときに，初期のリセット後にも‘X’がある場合には，不具合の可能性があるので，注意する必要があります．

VHDL

VHDLではフリップフロップの初期状態は，‘U’となります．これは未初期化や未定などと呼ばれ，値が定まっていない状態を表します．

VHDLのstd_logicには，‘1’，‘0’を含む9値の状態が用意されています．

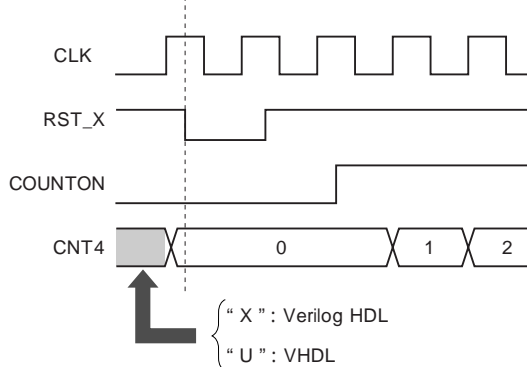
シミュレーション結果の波形を観察しているときに，初期のリセット後にも“U”がある場合には，不具合の可能性があるので，注意する必要があります．

図A リセット前のフリップフロップの値は決まらない
Verilog HDLでは不定状態を‘X’で表す．VHDLでは未初期化・未定状態を‘U’で表す．

図6　イネーブル付きの12進カウンタの状態の組み合わせ
可能性のある状態は，全部で48通りになる．

① 0から11までカウントした後，また0に戻ることを確認する．
② COUNTONが0のとき，本当にカウントが止まるか，CNT4の値が0から10までのどれかの場合と，11の場合を確認する．
③ RST_Xが0でカウント値がリセットされるか，COUNTONが0の場合と1の場合で確認する．

図7　検証仕様
最低限確認しなければならない項目に絞る．

ストベンチと同じ要領で作ることができます．

リスト5と**リスト6**は，**図8**のテスト入力をVerilog HDLとVHDLで記述したものです．

● クロック周期の工夫

リスト5と**リスト6**は，ともにクロックの周期を100nsとしていますが，すべての遅延を実際の数値で書いています．これではもし，クロック周期を10nsに変えてシミュレーションしたい場合，十数カ所すべてを書き換えなくてはいけません．しかし，Verilog HDL，VHDLともに数値を文字列に置き換える文法が用意されており，これを使えば周期の変更が非常に容易に行えます．

Verilog HDL

Verilog HDLではこれをパラメータという文法で実現します．**図9**はパラメータ宣言の書式です．

リスト5　Verilog HDLによるテスト入力の記述例

```
always begin
             CLK = 1'b1;
  #50        CLK = 1'b0;
  #50;
end

initial begin
  RST_X =1'b1; COUNTON = 1'b0;

  #10;

  #100       RST_X   = 1'b0;
  #100       RST_X   = 1'b1;
  #100       COUNTON = 1'b1;
  #1500      RST_X   = 1'b0;
  #100       RST_X   = 1'b1;
  #500       COUNTON = 1'b0;
  #100       COUNTON = 1'b1;
  #600       COUNTON = 1'b0;
  #200       RST_X   = 1'b0;
  #100       RST_X   = 1'b1;
  #100       COUNTON = 1'b1;
  #500       $finish();
end
```


リスト6　VHDLによるテスト入力の記述例

```
process begin
  CLK <= '1'; wait for 50 ns;
  CLK <= '0'; wait for 50 ns;
end process;

process begin
  RST_X <='1'; COUNTON <= '0';

  wait for 10 ns;

  wait for 100 ns;  RST_X   <= '0';
  wait for 100 ns;  RST_X   <= '1';
  wait for 100 ns;  COUNTON <= '1';
  wait for 1500 ns; RST_X   <= '0';
  wait for 100 ns;  RST_X   <= '1';
  wait for 500 ns;  COUNTON <= '0';
  wait for 100 ns;  COUNTON <= '1';
  wait for 600 ns;  COUNTON <= '0';
  wait for 200 ns;  RST_X   <= '0';
  wait for 100 ns;  RST_X   <= '1';
  wait for 100 ns;  COUNTON <= '1';
  wait for 500 ns;  assert false;
end process;
```
クロックの記述
レーシング対策

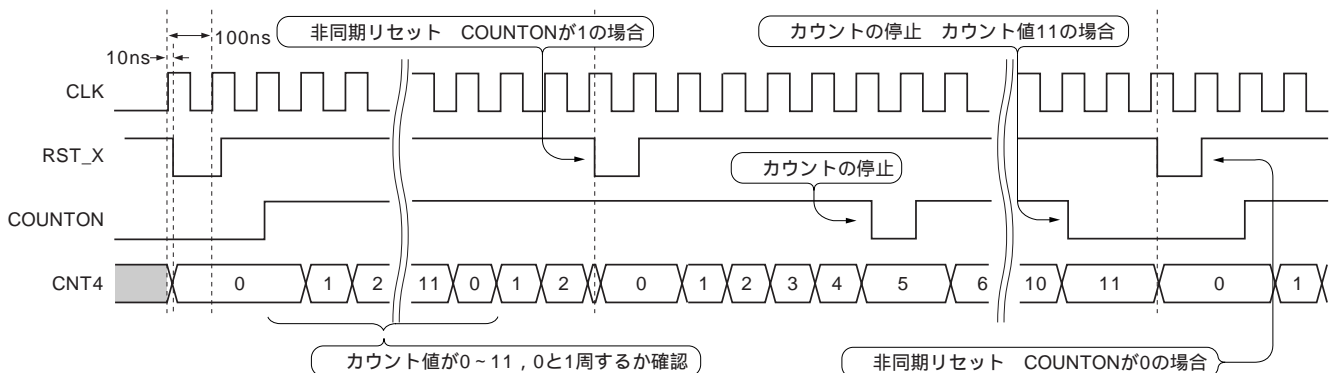



図8　イネーブル付きの12進カウンタのテスト入力
これだけの項目を見ることができれば，100％とはいえないまでも，ほとんどの不具合は発見できる．

図9のパラメータ名の部分には，initialやmoduleなどの予約語と重複しない限り，自由な文字列を使用できます．パラメータ宣言はmodule内で信号宣言と同じように宣言します．宣言されたパラメータは，その宣言が書かれたmodule内においてのみ有効で，設定した数値と同じように使うことができます．

コラム　検証仕様の洗い出し

本文では，48通りすべてを確認するのが大変なので，必要最低限のテストを抽出しているとしていますが，実際の開発では48通り程度で，すべての場合を確認できるのであれば確認します．しかし，実際の開発で検証される回路の規模はこの100倍，1000倍，もしくはそれ以上となり，考えられるすべての場合を確認しようとすると簡単に1万以上のバリエーションが生まれてしまいます．限られた開発期間の中では，これは現実的に実施不可能です．ですから検証項目の洗い出しは，各機能の要素の抽出と，その組み合わせをいかに必要最低限に抑えられるかが重要になります．

本文の**図7**に示した検証仕様の②では，カウント値が11の場合だけを特別扱いしています．これは次のような理由によります．

カウント値が0～10までは，COUNTONが1の場合は1が加算され，COUNTONが0の場合は同じ値を保持します．このカウント値が0～10まで動作を場合分けして回路を記述するとは考えられません．従って，カウント値が5のとき正しく動いていて，4や7のときだけ正しく動かない可能性は極めて低いと言えます．

しかし，カウント値が11の場合だけはほかの値のときとは動作が異なります．COUNTONが1の場合に，1が加算されるのではなく，カウント値が0に戻ります．回路の記述では，カウント値が11のときだけ別に条件を作っている可能性が高くなります．つまり，カウント値が0～10までのときに正しく動いたからと言って，11のときも正しく動くことは保障できません．

リストDと**リストE**は検証対象の回路の正しい設計例と，誤った設計例をVerilog HDLとVHDLで記述したものです．

誤った例ではカウント値が11のときに限って，COUNTONが0'でもカウントが止まらなくなります．このように機能の切り変わり目付近で不具合は発生しやすいので，検証では必ず機能の切り変わり目を確認する必要があります．

本文の**図8**では，カウントの停止をカウント値が11のとき以外では，0と5のときしか行われていません．100％の検証をめざすためには，必要最低限の項目を**図8**のように確認した後，時間の許す限り，万が一に備えてほかの値（1～4，6～10）も確認を続けていくべきです．これは，非同期リセットに関しても同様です．

リストD　Verilog HDLによる検証対象の回路の記述例

```
module counter(CLK,RST_X,COUNTON,CNT4);

input        CLK,RST_X,COUNTON;
output [3:0] CNT4;

reg    [3:0] CNT4;

always @(posedge CLK or negedge RST_X)
begin
  if( RST_X==1'b0 )
    CNT4 <= 4'b0000;
  else if( COUNTON==1'b1 )
    if(CNT4 == 4'b1011)
      CNT4 <= #1 4'b0000;

    else

      CNT4 <= #1 CNT4 + 1'b1;
end

endmodule
```

```
else if(CNT4 == 4'b1011)
       CNT4 <= #1 4'b0000;
else if( COUNTON==1'b1 )
       CNT4 <= #1 CNT4 + 1'b1;
```


リストE　VHDLによる検証対象の回路の記述例

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity counter is
   port (CLK,RST_X,COUNTON : in  std_logic;
         CNT4              : out std_logic_vector
                                  (3 downto 0));
end counter;

architecture RTL of counter is

  signal COUNT : std_logic_vector(3 downto 0);

begin

CNT4 <= COUNT;

process(CLK,RST_X)begin
  if(RST_X='0')then
    COUNT <= "0000";
  elsif(CLK'event and CLK='1')then

    if(COUNTON='1')then
      if(COUNT="1011")then
        COUNT <= "0000";

      else

        COUNT <= COUNT + '1' after 1 ns;
      end if;

    end if;
  end if;
end process;


end RTL;
```

```
if(COUNT="1011")then
  COUNT <= "0000";
elsif(COUNTON='1')then
  COUNT <= COUNT + '1' after 1 ns;
end if;
```

カウント値11のときの機能の記述

カウント値0から10のときの機能の記述

こんな誤りならカウント値11のときだけ誤動作

数値の部分には，数，もしくは式を書きます．パラメータを使ったテストベンチは，宣言部分の数値を書き換えるだけで，パラメータの書かれた部分の遅延値などをすべて変更できます．

リスト7はパラメータを使ったテストベンチの例です．パラメータCYCLEはクロック1周期，HALF_CYCLEはクロック半周期，DELAYはレーシング対策の遅延の値で，これらの変更はテスト入力内のすべてのパラメータに有効です（下掲のコラム「丸め精度について」を参照）．

VHDL

VHDLではこれを定数宣言という文法で実現します．**図10**は定数宣言の書式です．

図10の定数名の部分には，processやentityなどの予

```
parameter パタメータ名 = 数値;
```

図9　Verilog HDLによるパラメータ宣言の書式
パラメータ名の部分には，initialやmoduleなどの予約語と重複しない限り，自由な文字列を使用できる．

リスト7　Verilog HDLによるパラメータを使ったテストベンチ

```
module counter_tb();

parameter CYCLE      = 100;
parameter HALF_CYCLE = 50;      } ← パラメータ宣言
parameter DELAY      = 10;

reg        RST_X,CLK,COUNTON;
wire [3:0] CNT4;

counter counter(.CLK(CLK), .RST_X(RST_X),
                .COUNTON(COUNTON), .CNT4(CNT4));

always begin
                 CLK = 1'b1;
  #HALF_CYCLE CLK = 1'b0;
  #HALF_CYCLE;  ← パラメータの使用
end

initial begin
  RST_X =1'b1; COUNTON = 1'b0;
  #DELAY;  ← パラメータの使用
  #CYCLE       RST_X   = 1'b0;
  #CYCLE       RST_X   = 1'b1;
  #CYCLE       COUNTON = 1'b1;
  #(15*CYCLE)  RST_X   = 1'b0;
  #CYCLE       RST_X   = 1'b1;
  #(5*CYCLE)   COUNTON = 1'b0;
  #CYCLE       COUNTON = 1'b1;
  #(6*CYCLE)   COUNTON = 1'b0;
  #(2*CYCLE)   RST_X   = 1'b0;
  #CYCLE       RST_X   = 1'b1;
  #CYCLE       COUNTON = 1'b1;
  #(5*CYCLE)   $finish();
end

endmodule
```

コラム　丸め精度について

Verilog HDL　VHDL

リストF(a)と**リストG(a)**はクロックの半周期を，1周期のパラメータと割り算で記述した例です．しかし，このようにパラメータに割り算を使うべきではありません．もし割った後の値に端数があった場合，この丸め込みはシミュレータによって変わってしまう可能性があります．

丸め精度とは，シミュレーション時間をどこまでの精度で実現するかを設定する値です．丸め精度が1nsのとき，1ns以下の値は1ns単位に丸め込まれます．

例えば1周期を77nsとし，丸め精度が1nsの状態で**リストF(b)**と**リストG(b)**のように1周期のパラメータ・定数を2で割った場合，演算結果は38ns，または39nsとして扱われます．シミュレータによって結果が変わることを避けたい場合は，時間を設定するパラメータや定数に割り算を使うべきではありません．

リストF　Verilog HDLによるクロックの半周期の記述例

```
parameter CYCLE      = 100;
     ・・・
always begin
               CLK = 1'b1;
  #(CYCLE/2) CLK = 1'b0;      ×
  #(CYCLE/2);
end
```

(a) クロックの半周期を割り算で記述

```
parameter CYCLE      = 77;
     ・・・
always begin
               CLK = 1'b1;
  #(CYCLE/2) CLK = 1'b0;
  #(CYCLE/2);  ← 丸め精度が1nsなら
                 クロックの半周期は38nsか39ns
end
```

(b) 半周期がシミュレータによって異なる

リストG　VHDLによるクロックの半周期の記述例

```
constant CYCLE          : Time := 100 ns;
  ・・・
process begin
  CLK <= '1'; wait for CYCLE/2;
  CLK <= '0'; wait for CYCLE/2;      ×
end process;
```

(a) クロックの半周期を割り算で記述

```
  constant CYCLE          : Time := 77 ns;
  ・・・
process begin
  CLK <= '1'; wait for CYCLE/2;
  CLK <= '0'; wait for CYCLE/2;  ← 丸め精度が1nsなら
                                   クロックの半周期は
                                   38nsか39ns
end process;
```

(b) 半周期がシミュレータによって異なる

```
constant 定数名 : データ型 := 値;
```

図10 VHDLによる定数宣言の書式
定数名の部分には，processやentityなどの予約語と重複しない限り，自由な文字列を使用できる．

約語と重複しない限り，自由な文字列を使用できます．定数宣言はarchitectureの宣言部分内で信号宣言と同じように宣言します．宣言された定数は，その宣言が書かれたarchitecture内においてのみ有効で，設定した数値と同じように使うことができます．

データ型の部分には，今回遅延値の設定に使いたいので，Timeと書きます．Timeは時間の物理量で，数値＋時間単位を値として取ります(100 nsや50 psなど)．

値の部分には，今回データ型がTimeなので時間の物理量を書きます．定数を使ったテストベンチは，宣言部分の数値を書き換えるだけで，定数の書かれた部分の遅延値などをすべて変更できます．

リスト8は定数を使ったテストベンチの例です．定数CYCLEはクロック1周期，HALF_CYCLEはクロック半周期，DELAYはレーシング対策の遅延の値で，これの変更はテスト入力内のすべての定数に有効です(p.123のコラム「丸め精度について」を参照)．

4. 検証結果の確認

テストベンチが完成したら，シミュレーションを行います[編集部注1]．シミュレーション結果を波形で確認し，**図8**の通りに動作するかを確認してください．

前回はたった4状態だったので，一目で確認できたと思いますが，今回は数十サイクル(数十クロック周期)にも及ぶテストとなっているので，すべてのサイクルを一つ一つ毎回確認していくのは大変です．それでは，どうすればよいのかというと，確認必須のポイントを絞ります．

実はテストベンチを作成する際に洗い出した検証仕様の項目が，そのまま確認必須のポイントになります．

シミュレーション結果の波形が，検証仕様の項目を実現しており，かつ期待する結果になっているか(CNT4が11の後に0に戻っているかなど)を，確認してください．

編集部注1：無償で利用できるシミュレータを，本誌2007年3月号の付属DVD-ROMに収録している．

リスト8 VHDLによる定数を使ったテストベンチ

```
library IEEE;
use IEEE.std_logic_1164.all;

entity counter_tb is
end counter_tb;

architecture SIM of counter_tb is

component counter
   port (CLK,RST_X,COUNTON : in  std_logic;
         CNT4              : out std_logic_vector(3
downto 0));
end component;

  constant CYCLE            : Time := 100 ns;
  constant HALF_CYCLE       : Time :=  50 ns;   ← 定数宣言
  constant DELAY            : Time :=  10 ns;

  signal CLK,RST_X,COUNTON : std_logic;
  signal CNT4              : std_logic_vector(3
downto 0);

begin

ucounter : counter port map (
  CLK     => CLK     ,
  RST_X   => RST_X   ,
  COUNTON => COUNTON,
  CNT4    => CNT4    );

process begin
  CLK <= '1'; wait for HALF_CYCLE;   ← 定数の使用
  CLK <= '0'; wait for HALF_CYCLE;
end process;

process begin
  RST_X <='1'; COUNTON <= '0';
  wait for DELAY;
  wait for CYCLE;       RST_X   <= '0';
  wait for CYCLE;       RST_X   <= '1';
  wait for CYCLE;       COUNTON <= '1';
  wait for (15*CYCLE); RST_X   <= '0';
  wait for CYCLE;       RST_X   <= '1';
  wait for (5*CYCLE);  COUNTON <= '0';
  wait for CYCLE;       COUNTON <= '1';
  wait for (6*CYCLE);  COUNTON <= '0';
  wait for (2*CYCLE);  RST_X   <= '0';
  wait for CYCLE;       RST_X   <= '1';
  wait for CYCLE;       COUNTON <= '1';
  wait for (5*CYCLE);  assert false;
end process;

end SIM;

configuration cfg_counter_tb of counter_tb is
  for SIM
  end for;
end cfg_counter_tb;
```

やすおか・たかし
(株)エッチ・ディー・ラボ

<筆者プロフィール>
安岡 貴志．東京理科大学 理工学部 数学科卒業．前職のデザインセンターでは，3年間Verilog HDLによるASIC開発に携わる．2002年にエッチ・ディー・ラボに入社し，Verilog HDL，VHDL，SystemCによる開発に従事するほか，同社のトレーニング講師を務める．